Appendix A

```
/*
 * INET         An implementation of the TCP/IP protocol suite for the LINUX
 *              operating system.  INET is implemented using the  BSD Socket
 *              interface as the means of communication with the user level.
 *              IP/TCP/UDP checksumming routines.
 *              Code from tcp.c and ip.c.
 *              Free software; can be redistributed and/or
 *              modified under the terms of the GNU General Public License
 *              as published by the Free Software Foundation; either version
 *              2 of the License, or any later version.
 */

#include <asm/errno.h>

/*
 * computes a partial checksum, e.g. for TCP/UDP fragments
 */

/*
unsigned int csum_partial(const unsigned char * buff, int len, unsigned int sum)
*/

.text
.align 4
.globl csum_partial

          /*
           * In Ethernet and SLIP connections, buff
           * is aligned on either a 2-byte or 4-byte boundary.  Provides at
           * least a twofold speedup on 486 and Pentium if it is 4-byte aligned.
           * 2-byte alignment can be converted to 4-byte
           * alignment for the unrolled loop.
           */

csum_partial:
        pushl %esi
        pushl %ebx
        movl 20(%esp),%eax # Function arg: unsigned int sum
        movl 16(%esp),%ecx # Function arg: int len
        movl 12(%esp),%esi  # Function arg: unsigned char *buff
        testl $2, %esi          # Check alignment.
        jz 2f                   # Jump if alignment is ok.
        subl $2, %ecx           # Alignment uses up two bytes.
        jae 1f                  # Jump if we had at least two bytes.
        addl $2, %ecx           # ecx was < 2.  Deal with it.
```

```
        jmp 4f
1:      movw (%esi), %bx
        addl $2, %esi
        addw %bx, %ax
        adcl $0, %eax
2:
        movl %ecx, %edx
        shrl $5, %ecx
        jz 2f
        testl %esi, %esi
1:      movl (%esi), %ebx
        adcl %ebx, %eax
        movl 4(%esi), %ebx
        adcl %ebx, %eax
        movl 8(%esi), %ebx
        adcl %ebx, %eax
        movl 12(%esi), %ebx
        adcl %ebx, %eax
        movl 16(%esi), %ebx
        adcl %ebx, %eax
        movl 20(%esi), %ebx
        adcl %ebx, %eax
        movl 24(%esi), %ebx
        adcl %ebx, %eax
        movl 28(%esi), %ebx
        adcl %ebx, %eax
        lea 32(%esi), %esi
        dec %ecx
        jne 1b
        adcl $0, %eax
2:      movl %edx, %ecx
        andl $0x1c, %edx
        je 4f
        shrl $2, %edx           # This clears CF
3:      adcl (%esi), %eax
        lea 4(%esi), %esi
        dec %edx
        jne 3b
        adcl $0, %eax
4:      andl $3, %ecx
        jz 7f
        cmpl $2, %ecx
        jb 5f
        movw (%esi),%cx
        leal 2(%esi),%esi
        je 6f
        shll $16,%ecx
```

```
5:      movb (%esi),%cl
6:      addl %ecx,%eax
        adcl $0, %eax
7:
        popl %ebx
        popl %esi
        ret
```

Appendix B

```
// Code to reblock data segments as packets and calculate checksums for packets
//
// copyright 1999, Quantum Corp.
// author Rodney Van Meter

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

// #include "/usr/src/linux/include/linux/linkage.h"
// #include "/usr/src/linux/include/asm-i386/checksum.h"

// below code is based on the above two files in the Linux source code

/*
 * computes the checksum of a memory block at buff, length len,
 * and adds in "sum" (32-bit)
 *
 * returns a 32-bit number suitable for feeding into itself
 * or csum_tcpudp_magic
 *
 * this function must be called with even lengths, except
 * for the last fragment, which may be odd
 *
 * preferably buff is aligned on a 32-bit boundary
 */

unsigned int csum_partial(const unsigned char * buff, int len, unsigned int sum);

/*
 *      Fold a partial checksum
 */

static inline unsigned int csum_fold(unsigned int sum)
{
        __asm__("
                addl %1, %0
                adcl $0xffff, %0
                "
                : "=r" (sum)
                : "r" (sum << 16), "0" (sum & 0xffff0000)
        );
        return (~sum) >> 16;
}
```

```
// end of code that is based on said two files in the Linux source code


#define BUFSIZE 512
#define NUMBUFS 200
#define PKTSIZE 1460

#define MIN(a,b) ((a) < (b)) ? (a) : (b)

char buffer[BUFSIZE*NUMBUFS];

// checksums contains intermediate 32-bit values that
// need to be folded together and complemented before sending in TCP
unsigned int checksums[NUMBUFS];

unsigned int Checksum(unsigned char *bufp, int len);
unsigned int ChecksumAdd(unsigned int a, unsigned int b);
unsigned int ChecksumSubtract(unsigned int a, unsigned int b);

main(int argc, char *argv[])
{
 int fd;
 int BufferEntry = 0;
 int TotalRead = 0;
 int TotalSent = 0;
 int CachedFragment = -1;
 unsigned int CachedFragmentCksum;      // always inner fragment cksum kept
 int CFSize;                            /* how much of the buffer? */
 int offset = 0;
 unsigned int ThisCksum = 0;
 unsigned int retval;
 int i;

 if ((fd = open(argv[1],O_RDONLY)) < 0) {
  perror("right up front");
  exit(-1);
 }

 // basic two-pass operation;
 // first pass is to read all of the data into memory;
 // assumes the read always returns exactly the amount requested
 while (TotalRead < BUFSIZE*NUMBUFS) {
  retval = ReadWithChecksum(fd,           // file descriptor
                            &buffer[BufferEntry*BUFSIZE],   // buffer addr
                            BUFSIZE,                        // size of read
                            &checksums[BufferEntry]);               // put cksum here
```

```
  if (retval < 0) {
    perror("on read");
    exit(-1);
  }

  // add to our total
  TotalRead += retval;

  if (retval != BUFSIZE)
    break;
  BufferEntry++;
} // end of while loop


/*
 * The following performs data reblocking and segment checksum management,
 * in addition to the arithmetic for checksum addition
 * and subtraction.
 */

// second pass; now data is transmitted onto network
// using the checksums from above
// go just to the last full packet

while (TotalSent + PKTSIZE < TotalRead) {
  // build and send a single packet
  int PktSize;
  int bufno;
  int modulo;
  unsigned int tmpcs;
  int thisfragsize;

  PktSize = 0;
  ThisCksum = 0;

  // assumes use of the whole fragment; a packet
  // is never smaller than BUFSIZE (e.g.  for this code)
  // and always send a full packet
  while (PktSize < PKTSIZE) {
    bufno = (TotalSent + PktSize) / BUFSIZE;
    tmpcs = checksums[bufno];
    // modulo should be zero for every flag but the first
    modulo = (TotalSent + PktSize) % BUFSIZE;
    thisfragsize = MIN((PKTSIZE - PktSize), (BUFSIZE - modulo));

    if (modulo == 0 &&
        thisfragsize == BUFSIZE) {
```

```
        // whole (i.e. complete) data segment
        ThisCksum = ChecksumAdd(ThisCksum, tmpcs);
        printf("case a: whole segment checksum from hardware computed table\n");
    } else {
        //  only using a fragment, so calculate its cksum in software
        // and add it in, keeping the result cached in case the next
        // packet can use it
        if (CachedFragment == bufno) {
         // caching the single fragment won, now take advantage of it

         // this is equivalent to an error check
         if (CFSize != BUFSIZE - thisfragsize) {
          printf("Cannot use the cached fragment buf %d size %d for fragment size %d\n",
                 bufno,CFSize,thisfragsize);
          goto mustdofrag;
         }

         // was inner trailing for prior packet, outer leading for this one
         tmpcs = ChecksumSubtract(checksums[bufno],CachedFragmentCksum);

         ThisCksum = ChecksumAdd(ThisCksum, tmpcs);
         printf("case b: using cached fragment checksum from prior packet\n");

        } else {
        mustdofrag:
         // cannot use cached checksum, so calculate checksum

         if (thisfragsize < BUFSIZE / 2) {

          // cksum this fragment
          tmpcs = Checksum(&buffer[TotalSent + PktSize],thisfragsize);
          ThisCksum = ChecksumAdd(ThisCksum, tmpcs);
          printf("case c: checksumming this fragment\n");

         } else {

          // cksum the complementary fragment
          unsigned int tmpcsfrag,actualcs;

          int offset;
          if (modulo) {
           // leading, so back up to beginning of this buffer
           offset = TotalSent - modulo;
          } else {
           // trailing, so go out to the end of the pkt
           // and run to the end of the buffer
           offset = TotalSent + PktSize + thisfragsize;
```

```
            }

            tmpcsfrag = Checksum(&buffer[offset],
                                 BUFSIZE - thisfragsize);

            // this converts to the inner (needed) fragment cksum
            tmpcs = ChecksumSubtract(checksums[bufno],tmpcsfrag);

            ThisCksum = ChecksumAdd(ThisCksum, tmpcs);
            printf("case d: checksumming the complementary fragment\n");
          }

          // now cache the inner trailing fragment checksum
          CachedFragmentCksum = tmpcs;
          CachedFragment = bufno;
          CFSize = thisfragsize;
        }
      }

      PktSize += thisfragsize;

    } // end of while for checksum build

    SendPacket(&buffer[TotalSent],PKTSIZE,ThisCksum);
    TotalSent += PKTSIZE;
  } // end of while full packet's worth of buffer left

  // can additionally send final partial packet here;

  // dump the table
  for ( i = 0 ; i < NUMBUFS ; i++ ) {
    printf("%d: 0x%x\n",i,checksums[i]);
  }
}

int ReadWithChecksum(int fd,          /* file descriptor */
                     char *bufp, /* buffer pointer to fill */
                     int size,      /* how many bytes should be read? */
                     int *cksump /* pointer to where to put the checksum */
                     )
{
  /* return value is the number of bytes read.  Assumes read always returns
     what is requested for, unless it's the end of the file. */

  int retval;

  retval = read(fd,bufp,size);
```

```
  *cksump = Checksum(bufp,size);
  return retval;
}

unsigned int Checksum(unsigned char *bufp, int len)
{
  /* though not implemented here, a starting value can also be added */
  return csum_partial(bufp,len,0);
}

unsigned int ChecksumAdd(unsigned int a, unsigned int b)
{
  unsigned int retval;
  unsigned int carry;

  retval = a + b;
  carry = retval < a;
  retval += carry;

  return retval;
}

unsigned int ChecksumSubtract(unsigned int a, unsigned int b)
{
  // they are the same in ones-complement arithmetic
  unsigned int retval,tmp;

  retval = ChecksumAdd(a,~b);
  tmp = ChecksumAdd(retval,b);
  if (tmp != a) {
    printf("sub not sym? a: 0x%x b: 0x%x r: 0x%x r+b: 0x%x\n",
           a,b,retval,tmp);
  }
  return retval;
}

// this function prints out the 32-bit intermediate value
// calculated using the cached and logic circuit-generated checksums
// and double-checks it by recomputing the checksum on the whole packet
SendPacket(unsigned char *bufp, int len, unsigned int csum)
{
  unsigned int csum2;
  unsigned short folded,folded2;

  csum2 = Checksum(bufp,len);       /* double check */
  folded = csum_fold(csum);
  folded2 = csum_fold(csum2);
```

```
 if (csum != csum2)
   printf("*********** error!\n");
 printf("snd %d: cksum: 0x%x -> 0x%x, doublecheck: 0x%x -> 0x%x\n",
          len,csum,folded,csum2,folded2);
}
```